

Sarment Manual

Laurent GUÉGUEN

February 2, 2009

February 2, 2009

Contents

0.1	Data classes	4
0.1.1	class Sequence	4
0.1.2	class Matrice	6
0.2	List of data classes	10
0.2.1	class Lsequence	10
0.3	Count classes	11
0.3.1	class Compte	11
0.3.2	class Proportion	14
0.3.3	class Lcompte	17
0.3.4	class Lproportion	19
0.4	Partition classes	21
0.4.1	class Segment	21
0.4.2	class Partition	22
0.4.3	class Lpartition	25
0.4.4	class Parti_simp	26
0.5	Descripteur and Lexique classes	29
0.5.1	class Descripteur	29
0.5.2	class Lexique	31
0.5.3	Descriptors and Predictions	36

0.1 Data classes

0.1.1 class Sequence

module `sequence`

The documentation is here.

This class is used to store sequence data, such as genomic or proteic sequences. It represents a succession of letters, such as

`acagcaggcatagacaggatacagatttta.`

Positions in a `Sequence` are numbered from 0 to `length-1`.

A `Sequence` can have a name, which is written on the first line in fasta format, after the `>`.

Construction

`__init__` Optional keyword `fic` allows construction by reading a file.

`Sequence` is implemented as a tabular in C++. For construction, the memory for a `Sequence` must be allocated by:

`generate` generates an empty `Sequence` with a given length;

`read_nf` reads a filename in specific or FASTA format; to recognize the formats, a filename in specific format must end with `.seq`, and a filename in FASTA format must end with `.fa` or `.fst`;

`read_prop` builds or changes randomly from a `Proportion`; see `Sequence` generation.

Optional keywords:

`deb=d` changes only after position `d` (≥ 0) included;

`fin=f` changes only before position `f` ($< \text{len}()$) included;

`long=lg` creates a new `lg`-length `Sequence`. In that case, `deb` and `fin` are not read;

`read_Lprop` builds randomly from `Lproportion` and returns the resulting `Partition`; see `Sequence` generation.

Optional keywords:

`deb=d` changes only after position `d` (≥ 0) included;

`fin=f` changes only before position `f` ($< \text{len}()$) included;

`long=lg` creates a new `lg`-length `Sequence`. In that case, `deb` and `fin` are not read;

`etat_init=e` makes generation beginning with descriptor number `e` if it is valid. Otherwise, starts with a random descriptor of the `Lproportion`;

`read_Part` builds randomly from a `Partition` and a `Lproportion`; each `Segment` of the `Partition` must have descriptors numbers, and each number must be the number of a `Proportion` of the `Lproportion`. At each position, the `Proportion` corresponding to the number is used to randomly generate a letter, as for a `Sequence` generation;

`copy` copies deeply from another `Sequence`.

Handling

`__len__` returns the length of the `Sequence`;

`__getitem__` and `__getslice__` are implemented, to get respectively characters and sub-sequences.

Beware: operator `__getslice__` DOES NOT create a new `Sequence` object, but only a shallow copy, hence it must be used with care;

`__setitem__` is used to change a letter in the `Sequence`.

`__setslice__` is used to change a segment of the `Sequence` by the letters of a string or a `Sequence`. BEWARE: if the included part is of the same length as the replaced segment, the `Sequence` is modified in place, otherwise a new `Sequence` is built. Hence different behaviours can occur if the replacement is made inside a subsequence. See the example below.

For example:

```
> import sequence
> s=sequence.Sequence(fic="toto.fa")
> len(s)
10
> b=s[3:5]
> print b
3
ACG
> b[2]='A'
> print s[3:5]
3
ACA
> b[:2]="TT"
> print s[3:5] # b and s are still linked
3
```

```

TTA
> b[:2] = "ACG"
> print b
4
ACGA
> print s[3:5] # b and s are no longer linked
3
TTA

```

```
#####
```

`alpha` returns the list of different letters in the **Sequence**;

`shuffle` randomly shuffles the **Sequence** by $(\text{len} * (\log(\text{len}) + 1) / 2)$ random transpositions;

`g_name` sets the name;

`name` returns the name;

Input-Output

Specific format is:

description	example
length of the sequence	20
sequence with any spaces and returns as wanted	ACGGGAAGCTAA AGCTGCG T

`__str__` outputs in specific format;

`fasta` outputs in fasta format. The name of the **Sequence** is written on first line after ">"

Optional keyword:

`lg=d` sets the length of the lines (default: 80). If null, returns the sequence in one line.

`seq` outputs the mere sequence of letters as a **string**.

0.1.2 class **Matrice**

module `matrice`

The documentation is here.

This class is used to store sequences of vectors indexed by letters or characters numbers (ie numbers between 0 and 255 and prefixed by a #). For example, such vectors can be letters frequencies.

Via ascii code, there is equivalence between character numbers and letters.

Positions in a `Matrice` are numbered from 0 to length-1.

Construction

`__init__` Optional keyword `fic` allows construction by reading a file.

`Matrice` is a two-dimension tabular in C++. The `Matrice` format is described in Input-Output.

For construction, the memory for a `Matrice` must be allocated by either:

`generate` generates an empty `Matrice` of given length and given a list of letters and/or numbers (between 0 and 255);

`read_nf` reads a filename in specific format;

`copy` copies this `Matrice` into a NEW one;

`smoothe` smoothes a data either by summing or averaging the values over sliding windows of specified length and step-size. Uncomplete windows are not stored;

`prediction` computes at each position of a data the predictions of descriptors of a `Lexique`.

The numbers of the descriptors of the `Lexique` are set between 0 and 255, if needed.

`derivate` computes from a data the differences between successive positions;

`integrate` computes from a data cumulated sums from the first position;

`fb` in HMM context, uses Forward-Backward algorithm on a data using a `Lexique`.

The numbers of the descriptors of the `Lexique` are set between 0 and 255, if needed.

For each descriptor, at each position, the value set is the log-probability of the occurrence of this descriptor, given the HMM and the data.

`backward` in HMM context, uses Backward algorithm on a data using a `Lexique`.

The numbers of the descriptors of the `Lexique` are set between 0 and 255, if needed.

For each descriptor, at each position, the value is the log-probability of the

post-position part of the data, given the HMM and the descriptor at this position [Rab89];

forward in HMM context, uses Forward algorithm on a data using a Lexique.

The numbers of the descriptors of the **Lexique** are set between 0 and 255, if needed.

For each descriptor, at each position, the value is the log-probability of the ante-position part of the data and of the descriptor at this position, given the HMM [Rab89];

set_proba normalizes each line so that the values are the logarithms of probabilities. If the former values are $(x_i)_i$, the new ones are: $x_i - \log(\sum_i \exp(x_i))$;

exp replaces the values by their exponential;

ln replaces the values by their neperian logarithm;

shuffle randomly shuffles the **Matrice** by $(\text{len} * (\log(\text{len}) + 1) / 2)$ random transpositions;

Handling

__len__ returns the length of the **Matrice**;

n_desc returns the number of descriptors;

desc returns the list of the descriptors;

__getslice__ returns a sub-matrice.

Beware: this operator does NOT create a new **Matrice** object, but only a shallow copy, hence it must be used with care;

val returns the value on a letter at a position. The first argument of this function can be either a letter or a number. For example, **val('a',1)** is the same as **val(97,1)** and **val('#97',1)**;

g_val is used to change a value on a letter at a position in the **Matrice**. The second argument of this function can be either a letter or a number. For example, **g_val(0.5,'a',1)** is the same as **g_val(0.5,97,1)** and **g_val(0.5,'#97',1)**;

max returns the maximum value at a given descriptor.

For example:

```

>>> import matrice
>>> m=matrice.Matrice(fic="es.mat")
>>> print m
5
#20      B
0        1
2        1
3        4
1        4
5        1

>>> len(m)
5
>>> m.desc()
['#20', 'B']
>>> m.max(20)
5
>>> n=m[1:4]
>>> print n
3
#20      B
2        1
3        4
1        4

>>> n.val('#20',1)
3.0
>>> n.g_val(7,20,1)
>>> print n
3
#20      B
2        1
7        4
1        4

>>> print m
5
#20      B
0        1
2        1
1        4
1        4

```

```
#####
```

`--add--` returns a NEW **Matrice** which is the sum of corresponding values in both **Matrice**, if those **Matrice** have the same length and descriptors;

`--iadd--` adds to the values of the first **Matrice** the corresponding values from the second one, if both **Matrice** have the same length and descriptors;

`--sub--` returns a NEW **Matrice** which is the subtraction of corresponding values in both **Matrice**, if those **Matrices** have the same length and descriptors;

`--isub--` substrates from the values of the first **Matrice** the corresponding values from the second one, if both **Matrice** have the same length and descriptors.

`--imul--` multiplies the values of the **Matrice** by a given value.

`line` returns a dictionary which keys are the descriptors of the **Matrice** and corresponding items are the values at specified line;

Input-Output

Specific format is:

description	example
length of the Matrice	5
letters separated by spaces or tabulations	A C B
arrays of values separated by spaces or tabulations	3.09 4.5 3
	2 0 0
	1 0 0
	1.19302 2 5
	0 0.322 19.202

`--str--` outputs in specific format, in which columns are tabular separated.

0.2 List of data classes

0.2.1 class **Lsequence**

module `lsequence`

The documentation is here.

A **Lsequence** is simply a list of Sequences, and used for calculation of Lpartitions.

As it inherits of `list`, it has all the methods of `list`.

Construction

`__init__` Optional keyword `fic` allows construction by reading a file of Sequences in FASTA format;

`read_nf` appends to the list the Sequences in the file of given name, in FASTA format;

`read_Lprop` creates random Sequences from a `Lproportion`.

Optional keyword `etat_init` sets the descriptor number at first position of the sequence.

Input-Output

`__str__` returns the string of all Sequences in FASTA format.

0.3 Count classes

0.3.1 class `Compte`

module `compte`

The documentation is here.

This class is used to compute counts on word-occurrences, particularly inside sequences.

The special character `^` is used to represent beginnings and ends of sequences. Such a character is handy to build markovian models from a `Compte` (see `Proportion`).

Construction

`__init__` Optional keyword `fic` allows construction by reading from a filename in specific format;

`add_seq` adds to the count the words of a specified length that are in a sequence of letters. This sequence must have the operator `__getitem__`.

Optional keywords:

`deb=d` counts only after position `d` (≥ 0) included;

`fin=f` counts only before position `f` ($< \text{len}()$) included;

`alpha=a` counts only words which letters are in string `a`. If `a=""`, does not consider alpha;

`fact=f` each word counts `f` (default: 1).

`add_pseudo` adds to the count the given word with optional count (default value: 1).

- If an element of the list is a string, it adds this string with count 1.
- If an element of the list is a list `[s,c]` with `s` a string and `c` a number, the word `s` is added with count `c`.

`read_nf` builds from a filename in specific format;

Handling

`__getitem__` returns the count of the specified word. Character `.` is a wildcard for all letters, `^` excepted.

For example:

```
>>> import compte
>>> c=compte.Compte()
>>> c.add_seq("ABCBA",3)
>>> c.add_seq("BCBAA",3)
>>> c.add_seq("BABAB",3)
>>> print c
AA^      1
ABA      1
ABC      1
AB^      1
A^^      2
CBA      2
BAA      1
BAB      2
BA^      1
BCB      2
B^^      1
^AB      1
^BA      1
^BC      1

>>> c['A']
6
>>> c['A. '] # number of 'A's followed by a letter
4
>>> c['BAB']
2
>>> c['BCB']
2
>>> c['B.B']
```

```

4
>>> c [ 'B^' ]    # number of ending 'B's
1

#####

__iadd__ adds to self the counts of a Compte;

copy returns a new Compte which is the copy of self;

min returns a new Compte made of the minimum of the counts of self and another
    Compte;

max returns a new Compte made of the maximum of the counts of self and another
    Compte;

intersects returns a new Compte made of the counts of self that are counts of
    another Compte;

__idiv__ divides all of the counts by a specified value;

restrict_to returns a new Compte which is self restricted to the letters of the
    specified string. The end-character ^ is kept;

strip returns a new Compte from the words of self that do not have the end-
    character ^;

rstrip returns a new Compte from the words of self that do not end with char-
    acter ^;

lg_max returns the length of the longest word;

alph returns the list of the letters used in the counts;

words returns the list of the counted words;

prop returns the corresponding Proportion.

Optional keywords:

lpost=1 specifies the length of the posterior words, ie the words which fre-
    quencies are computed. Default: the maximum length of the words;

```

`lprior=1` specifies the length of the prior words, ie the words on which the computed words depend, in a markovian context. Default: 0;

As special character "`^`" stands for the limits of the sequence, the words terminating with this symbol are counted as " same length or longer than given length "-words;

`next` returns a list of [letter,count] of letters following the specified word;

`has_prefix` returns True if the specified word is a strict prefix of a word in the `Compte`;

`is_empty` returns True if the `Compte` is empty.

Input-Output

Specific format is:

description	example
lines of	AB 3
word and count separated by a space or a tabulation	B^ 5
	^BBC 1

`pref` returns the **string** of the `Compte` of words of specified length in same format as `__str__`;

`__str__` outputs in specific format.

0.3.2 class **Proportion**

module `compte`

The documentation is here.

This class is used for proportions of words that follow given words. For example, it can store the fact that the proportions of letters following word **AC** are:

```
A    0.34
C    0.15
G    0.23
T    0.28
```

Then a distinction is made between *prior* words (such as **AC** here), and *posterior* words (such as **A,C,G** and **T** here).

The special character `^` is used to represent beginnings and ends of sequences.

Construction

`__init__` Optional keywords:

`fic` allows construction by reading from a filename in specific format;

`str` allows construction by reading from a string in specific format;

`read_nf` builds from a filename in specific format;

`read_str` builds from a string in specific format;

`read_Compte` Builds from a `Compte`.

Optional keywords:

`lpost=1` specifies the length of the posterior words, ie the words which frequencies are computed. Default: the maximum length of the words;

`lprior=1` specifies the length of the prior words, ie the words on which the computed words depend, in a markovian context. Default: 0;

As special character "`^`" stands for the limits of the sequence, the words terminating with this symbol are counted as " same length or longer than given length "-words;

Handling

`__getitem__` returns the string, in format of `Compte` of the posterior corresponding to the given prior;

`limit_on` returns a SHALLOW copy of the `Proportion` limited to the priors that contain words of a given list;

`__iadd__` adds to `self` the proportions of another `Proportion`;

`KL_MC` computes Kullback-Leibler distance to a `Proportion`, by Monte Carlo simulation on several (default:100) `Sequence` of a given length (default:1000) generated by method `read_prop` of `Sequence`. See Sequence generation;

`lg_max` returns the length of the longest word, prior or posterior (prior+posterior);

`lg_max_prior` returns the length of the longest prior;

`lg_max_posterior` returns the length of the longest posterior;

`alph` returns the list of the letters used in the proportions;

`prefixes` returns the list of the prefixes;

`next` returns a list of [posterior,proportion] for the specified prior. The wild letter '.' takes for any the letters;

`rand_next` returns a posterior given a specified prior, randomly chosen among following the proportions of the prior;

`has_prefix` returns True if the specified word is a valid prior. Remember that character `^` stands for begin or end of sequence;

`is_empty` returns True if the `Proportion` is empty;

`has_post` returns True if the `Proportion` has a posterior with an empty prior.

Input-Output

Specific format is:

<u>description</u>	<u>example</u>
lines of	A B 0.3
prior posterior and count separated by a whitespace	A A 0.7 B B 0.5 B A 0.5 A 0.1 B 0.9 ~ A 0.5 ~ B 0.5

In this example, following an A, proportion of B is 0.3, and proportion of A is 0.7. Overall proportion of A is 0.1, and of B is 0.9. Proportion of beginning A is 0.5, as well as proportion of beginning B.

`__str__` outputs in specific format;

`loglex` returns the corresponding `Descripteur`. See `read_prop` in that class;

Sequence generation

From a `Proportion`, a (part of a) `Sequence` can be generated randomly, by the method `read_prop`.

The process is:

- for all increasing positions `i`:
 - get the longest word `w` ending in `i-1` that is a valid prior (using method `has_prefix`);
 - if there is a posterior corresponding to `w`, let `lp` be the list of corresponding couples [posterior,proportion] (using method `next`); otherwise, `lp` is the list of the uniform distribution of all letters of the `Proportion`;
 - randomly choose a posterior `p` according to the factors in list `lp` (see under);

- if the first letter of `p` is a terminating character (`^`), put character null (`'\0'`) at that position, and exit;
otherwise put that letter at position `i` on the **Sequence**.

Actually, the random choice of the posterior is made with probabilities proportional to their respective proportions, even if the sum of the proportions is different from 1. Then sequence generation is possible even with non-orthodox proportions.

0.3.3 class `Lcompte`

module `lcompte`

The documentation is here.

This class is like a dictionary in which the keys are the descriptor numbers, and the items the corresponding **Compte**.

Moreover, the number of transitions between such descriptors are stored. If not specified, this number of transitions is null.

The aim of this class is to build easily hidden Markov chains.

Construction

`__init__` Optional keyword `fic` allows construction by reading from a filename in specific format;

`read_nf` builds from a filename in specific format;

`read_nf_seq` counts specified-length words from a list of filenames. In each file are the counted sequences, and descriptor number (`n-1`) is related to file number `n`. Between-descriptors transitions are uniformly distributed.

Optional keyword:

`alpha=st` **Compte** are restricted to letters in string `st`. If `st=""`, behaves as if no option `alpha`.

`read_Lpart` counts the words of a specified length from a **Lpartition**. The **Compte** affected to the descriptor numbers are computed from the descriptors of the **Partitions** on the positions of the corresponding **Sequences**.

Optional keyword:

`alpha=st` **Compte** are restricted to letters in string `st`. If `st=""`, behaves as if not option `alpha`.

Handling

`strip` returns a new `Lcompte` from the words of `self` that do not have the end-character `^`, using `strip` method;

`rstrip` returns a new `Lcompte` from the words of `self` that do not end with character `^`, using `rstrip` method;

`__getitem__` returns the `Compte` of specified descriptor number;

`__setitem__` gets the specified `Compte` to the specified descriptor number;

`num` returns the list of descriptors numbers;

`lg_max` returns the length of the longest word;

`inter` returns the count of transitions between two descriptor numbers;

`g_inter` gets the count of transitions between two valid descriptor numbers;

`alph` returns the list of used letters;

`add_Partition` adds counts of words of specified length on the positions of a `Sequence`, on the basis of a `Compte` per descriptor number of a `Partition`;

Optional keyword:

`alpha=a` counts only words which letters are in string `a`. If `a=""`, as if not option `alpha`;

`fact=f` each word counts `f` (default: 1).

Input-Output

Specific format is:

description	example
sections of	1:
descriptor number:	AB 35
lines of word whitespace count	AA 71
	BB 55
lines of counts of transitions	BA 245
between descriptors numbers in format:	A~ 1
	B~ 9
num1ber, number2 whitespace count	
	2:
	ABB 45
	ABA 5
	AAA 24
	AAB 64
	AB 19
	A 1
	1,2 242
	1,1 854
	2,2 964
	2,1 610

`__str__` outputs in specific format.

0.3.4 class `Lproportion`

module `lcompte`

The documentation is here.

This class is like a dictionary in which the keys are the descriptor numbers, and the item the corresponding **Proportions**. Moreover, the probabilities of transitions between such descriptors are stored.

The aim of this class is to build easily hidden Markov chains.

Construction

`__init__` Optional keyword `fic` allows construction by reading from a filename in specific format;

`read_nf` builds from a filename in specific format;

`read_Lcompte` Builds from a `Lcompte`.

Optional keywords:

`lpost=1` specifies the maximum length of the posterior words, ie the words which frequencies are computed. Default: the maximum length of the words;

`lprior=1` specifies the length of the prior words, ie the words on which the computed words depend, in a markovian context. Default: 0;

Handling

`__getitem__` returns the **Proportion** of specified descriptor number;

`__setitem__` gets the specified **Proportion** to the specified descriptor number;

`num` returns the list of descriptors numbers;

`lg_max` returns the length of the longest word (prior+posterior);

`lg_max_prior` returns the length of the longest prior;

`lg_max_posterior` returns the length of the longest posterior;

`inter` returns the proportion of transitions between two descriptor numbers;

`g_inter` gets the proportion of transitions between two valid descriptor numbers;

`alph` returns the list of used letters;

`KL_MC` computes Kullback-Leibler divergence to a **Lproportion**, by Monte Carlo simulation on several (default:100) **Sequence** of a given length (default:1000) generated by method `read_Lprop` of **Sequence** . See Sequence generation;

Input-Output

Specific format is:

description	example
sections of	1:
decriptor number:	A B 0.3
lines of prior posterior whitespace count	A A 0.7
	B B 0.5
lines of probability transitions	B A 0.5
between descriptors numbers in format:	A 0.1
	B 0.9
number1, number2 whitespace proportion	
	2:
	AB B 0.5
	AB A 0.5
	AA A 0.4
	AA B 0.6
	A B 0.9
	A 0.1
	1,2 0.2
	1,1 0.8
	2,2 0.4
	2,1 0.6

`__str__` outputs in specific format;

`loglex` returns the corresponding Lexique. See `read_Lprop` in that class;

Sequence generation

From a `LProportion`, a (part of a) `Sequence` can be generated randomly, by the method `read_Lprop`.

The process is:

- for the first position, select a descriptor number for the initial state, either randomly or by the choice of the user. With that descriptor, select a letter at this position using the same process as `read_prop`, see `Sequence` generation.
- for all increasing positions `i`:
 - select a descriptor number given the former one, proportionnaly to between-descriptors transition probabilities;
 - with that descriptor, select a letter at this position using the same process as `read_prop`, see `Sequence` generation.

0.4 Partition classes

0.4.1 class Segment

module `segment`

The documentation is here.

Instances of this class are used for `Partition` class, but it may be useful to handle them.

A `Segment` is made of

- two positions (begin and end), *included* in the segment;
- a list of descriptors numbers (optionally empty);
- a float for the value of the segment (optionally zero).
- a string corresponding to a descriptor pattern (optionally this string is empty);

Construction

`__init__` Optional keywords:

`beg=d` begin position is `d` (default: 0);

`end=f` end position is `f` (default: 0);

`val=v` value is `v` (default:0).

`num=l` descriptor numbers are a copy of list `l` (default: []);

`dsc=s` descriptor is string `s` (default: "");

`read_str` builds from a string in specific format; returns `True` if parsing is ok, `False` otherwise;

`copy` returns a copy of this `Segment`.

Handling

`g_beg` gets the first position;

`g_end` gets the last position;

`g_num` gets the list of descriptors numbers;

`g_dsc` gets the string of the pattern;

`g_val` gets the value;

`deb` returns the first position;

`fin` returns the last position;

`num` returns the list of descriptors numbers;

`dsc` returns the string of the descriptor;

`val` returns the value;

`__len__` returns the length of the `Segment`, ie `fin-deb+1`.

Input-Output

Specific format is:

<p><u>description</u></p> <p><begin-end>sequence_of_descriptors_numbers:value:descriptor_pattern</p> <p>if sequence_of_descriptors_numbers is empty, it is not output</p> <p>if value is zero, it is not output</p> <p>if descriptor_pattern is empty, it is not output</p>
<p><u>examples</u></p> <p><0-123>1,2:-5.0:#{A(-1)CG}T</p> <p><5-3922>:0.45:</p> <p><87-332>1::</p>

`__str__` outputs in specific format;

`abr` outputs in specific format, without the descriptor pattern.

0.4.2 class `Partition`

module `partition`

The documentation is here.

Instances of this class are sets of `Segment` that part a `data` in several segments.

A *n*-partition is a partition with *n* segments.

A `Partition` is made of

- a list of `Segment`;
- a value, as much as possible the predictive value of the `Partition`;
- a name (the empty string if it does not have).

In all computations on data, if not specified, the first position of the partition is 0 and the last `len(data)-1`.

Construction

`__init__` Optional keyword `fic` allows construction by reading from a filename in specific format;

`s_name` sets the name from a given `string`;

`read_nf` builds from a filename in specific format;

`read_str` builds from a `string` in specific format;

`read_Matrice` builds from a `Matrice`, keeping at each position the descriptor number that is selected by a function. A `segment` is made for each run of identical descriptors numbers, and its value is the sum on its positions of the values returned by the function.

Optional keyword:

`func=f` uses function `f` for selecting the descriptor number. Function `f` has two arguments, a `Matrice` and a position, and returns a tuple descriptor number, floating point value (default: returns the tuple best descriptor, best value (the first of the bests descriptors is returned if there are several bests)).

`copy` builds a new `Partition` by copying this one;

`build_random` builds a random `Partition` on a given length with a given number of segments. Positions of the `segments` are uniformly distributed;

Optional keyword:

`ec=ec` sets the minimum length of the segments. It must be lower than the length of the sequence divided by (the number of segments +1) (default: 0).

`viterbi` using VITERBI algorithm (see [Rab89]), computes the most likely prediction `Partition` of a `Lexique` on a `Sequence`;

Optional keyword:

`maxseg=m` limits to `m` the maximum number of segments allowed in the computed partition (default: 10000). If `m` equals 0, there is no limit to this number.

`mpp` computes the maximum-prediction partition of a given number of segments by a `Lexique` on a `data`.

Handling

`__iadd__` appends a **Segment** after the highest position of the **Partition**;

`val` returns the value;

`name` returns the name;

`len_don` returns the data length;

`__len__` returns the number of **Segment**;

`num` returns the list of descriptors numbers;

`__getitem__` returns the **Segment** of a given number;

Other methods:

`group` returns a new **Partition** by clustering the **Segment** given their descriptors numbers. The argument is a list of numbers lists, each list being a set of clustered descriptors numbers. In the new **Partition**, the resulting **Segment** have no descriptors numbers.

Following the increasing positions order, the **Segment** are grouped as long as the set of the descriptors numbers of the group is included in a list of the argument; if this set is not included in such a list, a new **Segment** is built, and the new set is the descriptors numbers of the considered **Segment**;

`prediction` computes the prediction on a **data** by a **Lexique**, computing one best descriptor per class, without between descriptors transitions;

`pts_comm` on a **Partition**, it returns the number of positions where the descriptors numbers are the same in both **Partition**.

If the data-lengths are different, returns -1;

Input-Output

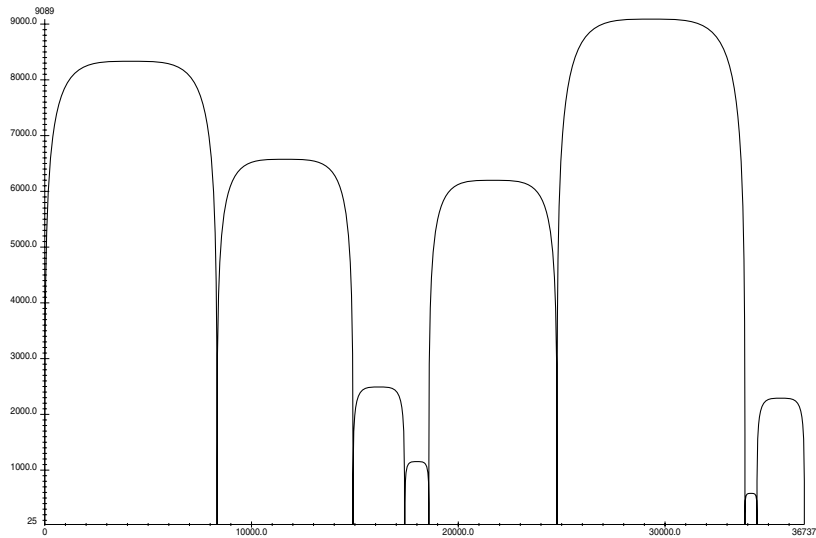
Specific format is:

<u>description</u> outputs of Segment separated by ' XXX ' —> value
<u>examples</u> <0-123>1,2:-5.0: +{A(-1)CG}T XXX <124-341>3:-7.0: ---> -12.0 <0-4>0:1.25: XXX <5-3922>:0.45: XXX <3923-4000>:0.31: ---> 2.01 <0-86>::: XXX <87-332>1::: ---> 0

`__str__` outputs in specific format;

`abr` outputs in specific format, without the descriptors patterns.

Graphical output



Horizontal axis represents the data, and each segment is drawn by an arc. The height of each arc is computed by a given function on the segments (here their lengths).

`draw_nf` outputs in postscript language in file of given name;

Optional keywords:

`seg=1` draws only segments which numbers are in list `1`;

`num=n` if equals 1, numbers of the descriptors are written;

`func=f` the height of each arc is proportional to value of function `f` computed on the corresponding `Segment`.

R language A drawing function of a `Parti_simp` in R language is available here.

0.4.3 class `Lpartition`

module `lpartition`

The documentation is here.

A `Lpartition` is simply a list of couples `[data, Partition]`, in which couples `partition` and `data` fit when the `Partition` exists.

As it inherits of `list`, it has all the methods of `list`.

Construction

`__init__`

`add_Lseq` appends to the data-list the list of `Lsequence`;

`add_don` appends to the data-list several copies of a `data`;

`build_random` creates random `Partition` on the data-list, using `build_random`;

`viterbi` computes the `Partition` on the stored `Sequence` with a `Lexique` using `viterbi` algorithm;

`fb` computes the `Partition` on the stored `Sequence` with a `Lexique` using forward-backward algorithm and `read_Matrice` method;

`mpp` computes a maximum-prediction `Partition` on the stored data with a `Lexique` using `mpp`;

Input-Output

`str_part` returns the string of the successive `Partition`, in their format.

0.4.4 class `Parti_simp`

module `parti_simp`

The documentation is here.

This class represents maximal predictive partitionings [Gué00, Gué01], ie a list of `Partition` of increasing number of segments, computed on a `data` using a `Lexique`.

A partitioning in n segments, or n -partitioning, is a list of partitions from 1 up to n segments classes.

A `Parti_simp` is made of

- a list of `Partition`;
- the value of the maximum prediction from a `Lexique` on a `data`; if this value is not defined, default value is 0;
- the minimum number of classes necessary to get the maximum prediction; if this maximum prediction value is not defined, default value is 0.

Construction

`__init__` Optional keyword `fic` allows construction by reading from a filename in specific format;

`read_nf` builds from a filename in specific format; if a `Partition` in the file has no name, its new name is the `string` of its index in the list;

`build_random` builds a random partitioning in a given number of classes on a given data-length, using `build_random`;

`mpp` computes the maximal predictive partitioning of a given number of classes with `Lexique` on a data; the names of the `Partitions` are their numbers of segments;

Handling

`len_don` returns the length of the data of the first `Partition`;

`__len__` returns the number of `Partition`;

`__getitem__` returns the `Partition` of a given number;

`__delitem__` removes the `Partition` of a given number;

`__getslice__` returns a `Parti_simp` made of the selected partitions;

`__delslice__` removes the `Partition` of indexes between given numbers;

`append` appends a `Partition` to the end of self. Data length of this `Partition` must be equal to the one of self; if this new `Partition` has no name, it gets the length of the `Parti_simp` as name, using method `s_name`.

`insert` inserts a `Partition` before given index. Data length of this `Partition` must be equal to the one of self; if this new `Partition` has no name, it gets the length of the `Parti_simp` as name, using method `s_name`.

`filter` returns a `Parti_simp` made of the partitions on which a given function returns `True`;

`group` returns a new `Parti_simp` by clustering the `Segment` of its `Partition` given their descriptors numbers, using `group`.

`ls_val` returns the list of the values of the `Partition`, using `val`;

`prediction` computes the list of predictions of the partitions on a `data` by a `Lexique`, using `prediction`;

`pts_comm` returns, for each number n of segments, the number of same-descriptor positions between the n -partitions of both `Parti_simp`, using `pts_comm`;

Input-Output

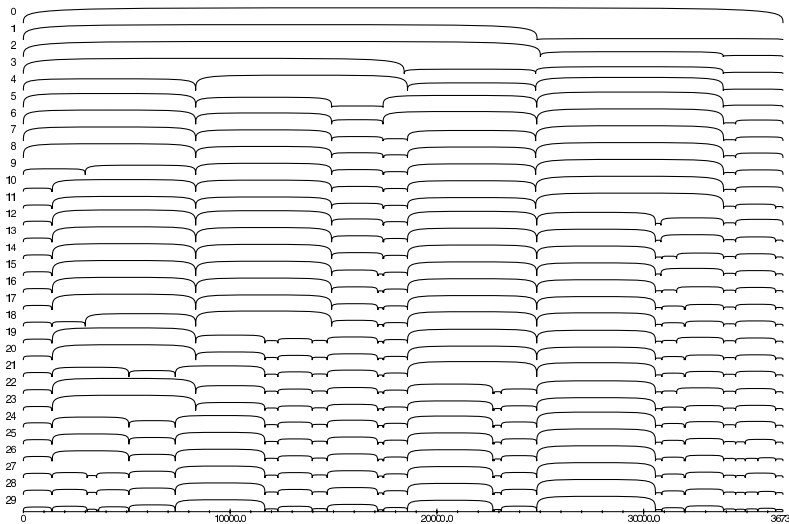
Specific format is:

<u>description</u> lines of outputs of Partition in specific format MAX(min-max number of classes) ---> maximum value
<u>example</u> <0-3637>6:-9127: ---> -9127 <0-2845>7:-6194: XXX <2846-3637>3:-2920: ---> -9114 <0-2505>7:-6234: XXX <2506-3349>1:-2139: XXX <3350-3637>51:-721: ---> -9094 MAX(366) ---> -8685

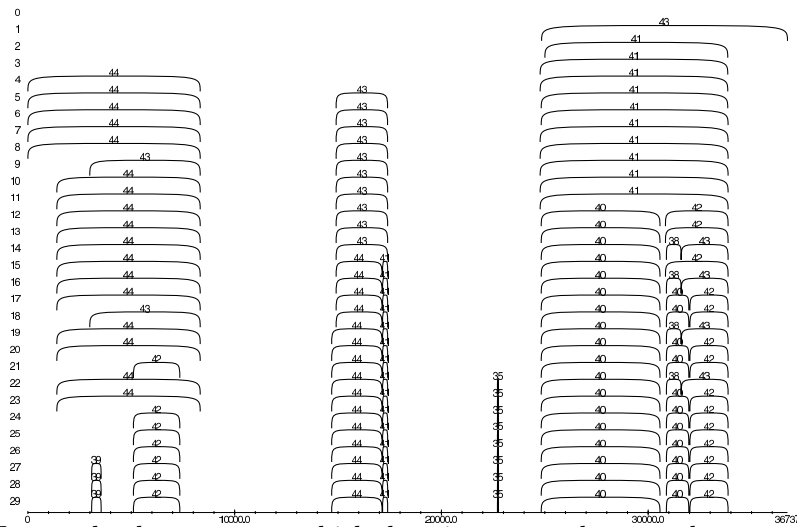
`__str__` outputs in specific format;

`abr` outputs in specific format, without the descriptors patterns (see `abr`).

Graphical output



Horizontal axis represents the data. On each line is the number of segments and the graphical output of the corresponding **Partition**, where each segment is drawn by an arc. The height of each arc is optionally computed by a given function on the segments (here their lengths).



Here, only the segments which descriptors numbers are between 30 and 44 are drawn, and their descriptors numbers are written above them.

`draw_nf` draws in postscript language in file of given name;

Optional keywords:

`seg=1` draws only segments which numbers are in list 1;

`num=n` if equals 1, the numbers of the descriptors are written;

`func=f` the height of each arc is proportional to value of function `f` computed on the corresponding `Segment`.

R language A drawing function in R language is available here.

0.5 Descripteur and Lexique classes

0.5.1 class Descripteur

module `descripteur`

The documentation is here.

Instances of this class correspond to simple descriptors to which numbers are set.

Construction

`__init__` builds a `Descripteur` with a given number.

Optional keyword:

`str=s` builds from string `s`, using `read_str`;

`prop=p` builds from `Proportion p`, using `read_prop`;

`fic=f` builds from filename `f`, using `read_nf`;

`fprop=f` builds from filename `f` of `Proportion`, using `read_prop`;

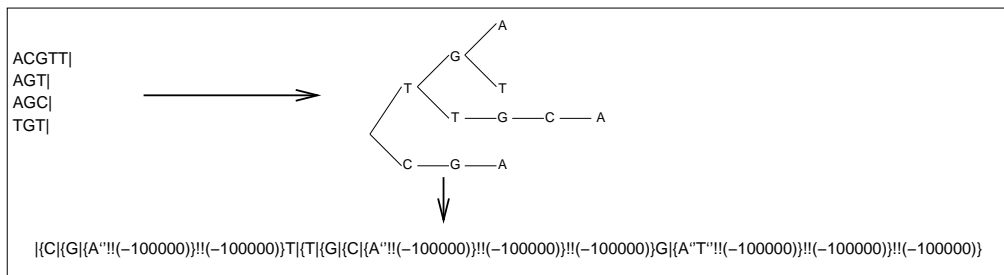
`read_str` builds from a string, in format of simple descriptors;

`read_nf` builds from a filename;

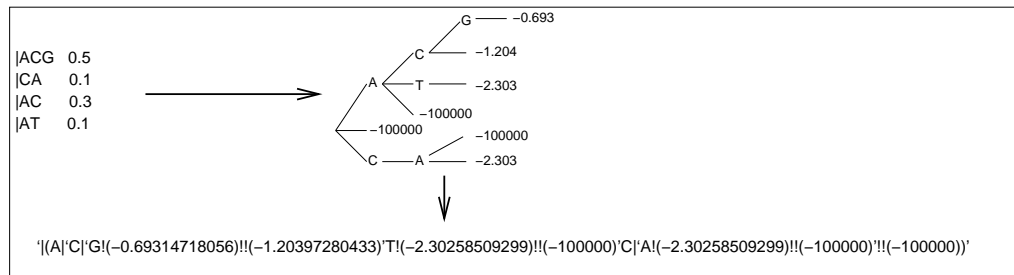
`read_prop` builds from a `Proportion`, using `loglex`. For efficiency considerations, the proportions priors and posteriors are translated differently (see descriptors):

- priors are clustered in a tree from the last letters to the first ones, and each leaf of this tree will be the output of the corresponding posterior. Afterwards, these labels are output with backward-or (`|{}`) operator along the edges of the tree, and the posteriors are output in their specific format. The order in backward-or operator is not related with any order in the proportions. Priority is given to the most specific prior. The `-100000` labels are for no-matching priors.

Example: The output of the posteriors are replaced by the ‘’.



- posteriors are clustered in a tree from the first letters to the last ones, and each leaf of this tree is the logarithm of the corresponding proportion. Afterwards, the labels are output with the here-or (`|()`) and forward-or (`|‘`) operators. `-100000` labels are for not-matching posteriors.



`copy` returns itself deeply copied `Descripteur`, with a given number;
`equals` copies from another `Descripteur`.

Handling

`num` returns the number of the `Descripteur`.

`prediction` computes the prediction on a `data`, at a given position.

Input-Output

`Descripteur` instances have a specific format that is described in Simple descriptors section.

`__str__` outputs in specific format.

0.5.2 class `Lexique`

module `lexique`

This class is used for all of the computations of partitions, partitionings and HMM analysis.

A lexique is a set of descriptors. The prediction functions, used for all the computations, are defined by the descriptors.

Beware: The descriptors #33 and #94 should not be used (see descriptors).

Optionally, some values can be put for the transitions between the descriptors: these values are used for HMM computations.

Construction

`__init__` Optional keywords:

`str=s` builds from string `s`, using `read_str`;

`alpha=a` uses the letters of string `a`, in use with option `str` (see input);

`Lprop=l` builds from `Lproportion l`, using `read_Lprop`;
`fic=f` builds from filename `f`, using `read_nf`;
`fprop=f` builds from filename `f` of `Lproportion`, using `read_Lprop`;
`read_nf` builds from a filename and optionally a string of letters (see input);
`read_str` builds from a string and optionally another string of letters (see input).
`read_Lprop` builds from a `Lproportion`, each `Proportion` being translated as in `read_prop`, and the transitions between descriptors are valued by the logarithms of the transitions proportions.

Handling

`is_empty` returns `True` if there is no descriptor, `False` otherwise;
`__len__` returns the number of descriptors;
`ls_num` returns the list of the numbers of the descriptors;
`met_au_net` removes repetitions in patterns of descriptors (like `ACAC` cleaned in `AC`) and repetitions of descriptors (like `A A` cleaned in `A`);
`__iter__` iterates over the descriptors, using `for d in lx`;
`__delitem__` removes the `Descripteur` of given number;
`__getitem__` gets the `Descripteur` of given number by `COPYING` it;
`init_trans` initialises the array of the transition probabilities;
`g_trans` gets the logarithm of the transition probability between `Descripteurs` of given numbers;
`__iadd__` add the copies of the `Descripteur` of a given `Lexique`. If the number of a `Descripteur` is already in the `Lexique`, this `Descripteur` is not added;
`__setitem__`

- either sets the `Descripteur` of given number to given `Descripteur` by `COPYING` it (if the number is already used in the `Lexique`, the corresponding descriptor is replaced by the new one);
- or builds a descriptor pattern given a tuple of numbers (numbers that must not have been already used in the `Lexique`), and a tuple or a list of `Descripteurs`.

For example:

```

>>> import lexique
>>> l=lexique.Lexique(str="1:A_2,3:BC")
>>> print l
3,2:CB 1:A

>>> import descripteur
>>> d=descripteur.Descripteur(3,str="Z")
>>> print d
Z
>>> l[2]=d
>>> print l
3,2:CZ 1:A

>>> d.read_str("P")
>>> print l
3,2:CZ 1:A

>>> e=descripteur.Descripteur(3,str="Y")
>>> l[5]=e
5:Y
>>> print l
5:Y      3,2:CZ 1:A

>>> l[4,3]=e,d
Bad descriptor number 3 already used
>>> l[4,6]=e,d
4,6:YP
>>> print l
6,4:PY 5:Y      3,2:CZ 1:A

>>> del l[3]
>>> del l[2]
>>> print l
6,4:PY 5:Y      1:A

>>> l[3,2]="X",e
3,2:XY
>>> print l
2,3:YX 6,4:PY 5:Y      1:A

#####

```

prediction computes the maximum prediction on a **data**, *without* using between-descriptors transitions;

Optional keywords:

deb=d sets the first position of the segment on which the prediction is computed (default: 0);

fin=f sets the last position of the segment on which the prediction is computed (default: last position-1);

If **deb>fin**, it returns 0.

val_max computes the maximum sum of predictions on a **data**, *without* using between-descriptors transitions; in that case, the maximal prediction is computed on each position, and the sum of these values is returned;

Optional keywords:

deb=d sets the first position of the segment on which the prediction is computed (default: 0);

fin=f sets the last position of the segment on which the prediction is computed (default: last position-1);

If **deb>fin**, it returns 0.

llh computes the best log-likelihood on a **Sequence**.

Optional keywords:

deb=d sets the first position of the segment on which the log-likelihood is computed (default: 0);

fin=f sets the last position of the segment on which the log-likelihood is computed (default: last position-1);

If **deb>fin**, it returns 0.

ls_value returns the dictionary of tuple of (the pattern of) numbers of descriptors, prediction of this (pattern of) descriptor on the data, for all descriptors, on a given **data**.

Optional keywords:

deb=d sets the first position of the segment on which the predictions are computed (default: 0);

fin=f sets the last position of the segment on which the predictions are computed (default: last position -1);

If **deb>fin**, it returns 0.

`windows` computes the list of the best predictions on the data in a sliding window of a given size and with steps of a given length;

`log_likelihood` computes the list of the log-likelihoods of the data given the numbers of segment, up to a given number of segments.

Input-Output

- Output

Specific format is:

```
description
numbers_of_descriptors_separated_by_commas:
    outputs_of_the_descriptors_in_the_pattern_separated by spaces.

optional lines of transitions costs between the descriptors, in format:
number_of_descriptor,number_of_descriptor    cost
```

examples

```
1,2:#{A(-1)CG}T 3:C
```

```
1:A 2:C 3:G 4:T
```

```
1,2,3:A|'C(0.3)GT(0.2)A'{C(0.5)A(-1.5)} 5:+(CG) 10:'T(-0.23)!(0.1)'
```

```
1:A 2:T
```

```
1,1 -2.3
```

```
1,2 -4.1
```

```
2,2 7.2
```

```
2,1 0
```

`__str__` outputs in specific format.

- Input

Input is in the same format as the output, plus:

- there is no need to give numbers to the descriptors. In that case, they will be automatically numbered;

- character `$` is a wildcard, and cannot be used as a standard letter. It is used in method `read_str`, and is set successively on all of the letters of the second string argument of this method. For example:
 - * `lit_str("$", "ACG")` builds Lexique: A C G
 - * `lit_str("A$A", "TG")` builds Lexique: ATA AGA
 - * `lit_str("A$$", "CGT")` builds Lexique:
 `'ACC' 'ACG' 'ACT' 'AGC' 'AGG' 'AGT' 'ATC' 'ATG' 'ATT'`

0.5.3 Descriptors and Predictions

Simple descriptors

A simple descriptor can be seen as a function applied to a position in a `data` and its vicinity, and returning a floating-point value.

In a `data`, on a position, a letter has a value:

- in a `Sequence`, the value of the existing letter is 1, the value of other letters is 0;
- in a `Matrice`, the value of an existing letter is the value in the data, the value of the other letters is 0.

A floating-value written between parentheses after a descriptor multiplies the prediction of this descriptor by that value. For example, on a `Sequence`, descriptor `A` returns 1 on `A`, and 0 elsewhere, whereas descriptor `A(0.7)` returns 0.7 on `A`, and 0 elsewhere.

For operators, notation is a prefix one.

The accepted descriptors are:

letters for letters between `a` and `z` and between `A` and `Z`, returns the value of the corresponding letter in the data.

```
letter ::= "a"... "z" | "A"... "Z"
```

special characters

- ! returns 1 in any position (even if out of bounds);
- ^ returns 1 if the position is out of bounds, 0 otherwise.

```
special ::= "^" | "!"
```

character codes for numbers between 0 and 255 included. Character codes of letters are output as letters.

Beware: As the codes of special characters `!` and `^` are 33 and 94, these codes must be used very cautiously.

```
character ::= #0..255
```

here-plus returns the sum of the predictions of the descriptors between the parentheses, at this position ; for example `+(ABC)`.

`here-plus ::= +(descriptors)`

here-mult returns the product of the predictions of the descriptors between the parentheses, at this position ; for example `*(ABC)`.

`here-mult ::= *(descriptors)`

here-or returns the prediction on the *current* position of a descriptor (the computing descriptor) chosen by the positivity of the prediction of another descriptor (the testing descriptor) on this position. Each couple testing descriptor-computing descriptor is written in this order. Between the brackets, the tests are made from left to right in the odd descriptors, and stop at the *first* positive test.

For example, on position 0 of Sequence ABC, prediction of

`|('AB'A(0.1)'AC'A(0.2)'AA'A(0.3))` returns 0.1.

For example, on position 0 of Sequence ABC, prediction of

`|('AB'A(0.1)'AB'A(0.2))` returns 0.1.

`here-or ::= |(descriptorsdescriptors)`

forward returns the prediction at the current position of the first descriptor between the quotes if the predictions of the next descriptors on the following positions are all positive.

For example, on position 0 of Sequence ACBS, prediction of

`'A(0.5)CB(0.3)'` returns 0.5.

`forward ::= 'descriptors'`

forward-or returns the prediction on the *current* position of a descriptor (the computing descriptor) chosen by the positivity of the prediction of another descriptor (the testing descriptor) on the *next* position. Each couple testing descriptor-computing descriptor is written in this order. Between the brackets, the tests are made from left to right in the odd descriptors, and stop at the *first* positive test.

For example, on position 2 of Sequence ACB, prediction of

`|'BC(0.1)CC(0.2)AC(0.3)'` returns 0.1.

For example, on position 2 of Sequence ACB, prediction of

`|'BC(-0.1)BC(0.2)'` returns -0.1 .

When the computing descriptor is an "or"-operator (here-or, backward-or, or forward-or), the current position for the tests inside this computing descriptor is the preceding preceding. Yet, their joined computing descriptors are used on the actual current position.

For example, on position 1 of Sequence CAB, prediction of
`| 'B| 'BC(0.1)CC(0.2)AC(0.3)'C| 'BC(0.4)CC(0.5)AC(0.6)'A| 'BC(0.7)CC(0.8)AC(0.9)''`
returns 0.7.

`forward-or ::= |'descriptorsdescriptors'`

backward returns the prediction at the current position of the last descriptor between the brackets if the predictions of the previous descriptors on the preceding positions are all positive.

For example, on position 3 of Sequence ACBS, prediction of
`{A(0.5)CB(0.3)}` returns 0.3.

`backward ::= {descriptors}`

backward-plus returns the sum of the predictions of the descriptors between the brackets, the last descriptor being applied on the current position, the preceding one on the position before, and so on.

For example, on position 4 of Sequence DABC prediction of
`+{A(0.5)B(-0.2)C(1.8)}` returns 2.1.

`backward-plus ::= +{descriptors}`

backward-or returns the prediction on the *current* position of a descriptor (the computing descriptor) chosen by the positivity of the prediction of another descriptor (the testing descriptor) on the *preceding* position. Each couple testing descriptor-computing descriptor is written in this order. Between the brackets, the tests are made from left to right in the odd descriptors, and stop at the *first* positive test.

For example, on position 3 of Sequence ABC, prediction of
`|{BC(0.1)CC(0.2)AC(0.3)}` returns 0.1.

For example, on position 3 of Sequence ABC, prediction of
`|{BC(-0.1)BC(0.2)}` returns -0.1.

When the computing descriptor is an "or"-operator (here-or, backward-or, or forward-or), the current position for the tests inside this computing descriptor is the preceding preceding. Yet, their joined computing descriptors are used on the actual current position.

For example, on position 3 of Sequence ABC, prediction of
`|{B|{BC(0.1)CC(0.2)AC(0.3)}C|{BC(0.4)CC(0.5)AC(0.6)}A|{BC(0.7)CC(0.8)AC(0.9)}}`
returns 0.3.

`backward-or ::= |{descriptorsdescriptors}`

Nb: these descriptors have been built for specific needs (such as traduction of markovian transition probabilities) but, owing to the C++ implementation, it is very easy to conceive new ones if necessary.

Descriptors patterns

A pattern of descriptors is used in the context of maximum predictive partitioning. It is a word of successive simple descriptors, used periodically to compute predictions on **data**. The period starts with the first descriptor on the first position.

For example, as the prediction on a data is the sum of the predictions on all the positions of the data, the prediction on sequence **ACBCAB** of descriptor pattern

AC is 4,

and prediction of descriptor pattern

CA is 0.

Prediction

On a position, the prediction value is the value of the used descriptor.

On a **data**, the prediction of a simple descriptor is the sum of the predictions on all of the positions of the data.

In the case of a descriptor pattern, the descriptors are used periodically, starting with the first descriptor of the pattern at the first position.

For example, on sequence **ACBCAB** the prediction of descriptor pattern

AC is 4,

and prediction of descriptor pattern

CA is 0.

Inside a **Lexique**, when there are transition-costs between descriptors, these costs are used in HMM context, ie in methods **fb**, **backward**, **forward**, and **viterbi**. In that case, these costs are added to the prediction at each transition between the descriptors.

Bibliography

- [Gué00] L. Guéguen. *Partitionnement maximalement prédictif sous contrainte d'ordre total. Applications aux séquences génétiques*. Thèse, Université Pierre et Marie CURIE - Paris VI, janvier 2000.
- [Gué01] L. Guéguen. Segmentation by maximal predictive partitioning according to composition biases. In O. Gascuel and M.F. Sagot, editors, *Computational Biology*, volume 2066 of *LNCS*, pages 32–45. JOBIM, May 2000 2001.
- [Rab89] L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Proc. IEEE*, volume 77, pages 257–285, 1989.